加餐-函数式编程拾遗

你好，我是郑晔！

我们之前用了三讲的篇幅讲了函数式编程，相信函数式编程在你心目中已经没有那么神秘了。我一直很偏执地认为，想要成为一个优秀的程序员，函数式编程是一定要学习的，它简直是一个待人发掘的宝库，因为里面的好东西太多了。

不过，考虑到整个课程的主线，我主要选择了函数式编程在设计上有较大影响的组合性和不变性来讲。但其实，函数式编程中有一些内容，虽然不一定是在设计上影响那么大，但作为一种编程技巧，也是非常值得我们去了解的。

所以，我准备了这次加餐，从函数式编程再找出一些内容来，让你来了解一下。相信我，即便你用的不是函数式编程语言，这些内容对你也是很有帮助的。

好，我们出发！

惰性求值

还记得我们第17讲的那个学生的例子吗？我们继续使用学生这个类。这次简化一点，我只使用其中的几个字段：

```
class Student {
  // 学生姓名
  private String name;
  // 年龄
  private long age;
  // 性别
  private Gender gender;

  public Student(final String name,
                 final long age,
                 final Gender gender) {
    this.name = name;
    this.age = age;
    this.gender = gender;
  }
}
```

然后，我们来看一段代码，你先猜猜这段代码的执行结果会是什么样子：

```
// 数据准备
Student jack = new Student("Jack", 18, Gender.MALE);
Student rose = new Student("Rose", 18, Gender.FEMALE);
List<Person> students = asList(jack, rose);

// 模拟对象
Function<Person, String> function = mock(Function.class);
when(function.apply(jack)).thenReturn("Jack");
```

```
// 映射
students.stream().map(function);

// 验证
verify(function).apply(jack);
```

这段代码里，我们用到了一个mock框架mockito，核心就是验证这里的function变量是否得到了正确的调用，这其中就用到了我们在第18讲中提到的map函数。

也许你已经猜到了，虽然按照普通的Java代码执行逻辑，verify的结果一定是function得到了正常的调用，但实际上，这里的function并没有调用。也就是说，虽然看上去map函数执行了，但并没有调用到function的apply方法。你可以试着执行这段代码去验证一下。

为什么会是这样呢？答案就在于这段代码是惰性求值的。

什么叫惰性求值呢？**惰性求值（Lazy Evaluation）是一种求值策略，它将求值的过程延迟到真正需要这个值的时候**。惰性求值的好处就在于可以规避一些不必要的计算，尤其是规模比较大，或是运行时间比较长的计算。

其实，如果你学习过设计模式，惰性求值这个概念你应该并不陌生。有一些设计模式就是典型的惰性求值，比如，Proxy模式，它就是采用了惰性求值的策略，把一些消耗很大的计算延迟到不得不算的时候去做。还有Singleton模式有时也会采用惰性求值的策略，在第一次访问的时候，再去生成对象。

在函数式编程中，惰性求值是一种很常见的求值策略，也是因为惰性求值的存在，我们可以做出很多有趣的事情。

无限流

在传统的编程方式中，我们熟悉的集合类都是有限长度的，因为集合中的每个元素都是事先计算好的。但现在有了惰性求值，我们就可以创造出一个无限长的集合。

你可能会有疑问，怎么可能会有无限长的集合呢？内存也存不下啊？如果你这么想的话，说明你的思路还是传统的方式。无限长集合中的元素并不是预置进去的，而是在需要的时候，才计算出来的。

无限长集合真正预置进去的是，元素的产生规则。这样一来，元素就会像流水一样源源不断地产生出来，我们将这种集合称为无限流（Infinite Stream）。

比如，我们要产生一个自然数的集合，可以这么做：

```
Stream.iterate(1, number -> number + 1)
```

在这里，我们定义了这个集合的第一个元素，然后给出了后续元素的推导规则，一个无限流就产生了。

当然，因为惰性求值的存在，这么定义的一个无限流并不会做真正的计算，只有在我们需要用到其中的一些

元素时，计算才会执行。比如，我们可以按需取出一些元素，在下面这段代码中，我们跳过了无限流的前两个元素，然后，取出三个元素，将结果打印了出来：

```
Stream.iterate(0, number -> number + 1)
  .skip(2)
  .limit(3)
  .forEach(System.out::println);
```

也许你会关心，什么情况下无限流才会真正的求值呢？其实，我们前面讲组合性时提到过，有一些基础的列表操作，列表操作可以分为两类，中间操作（Intermediate Operation）和终结操作（Terminal Operation），像map和filter这一类的就是中间操作，而像reduce一类的就属于终结操作。只有终结操作才需要我们给出一个结果，所以，只有终结操作才会引起真正的计算。

你可能会好奇，无限流的概念很有意思，但它有什么用呢？如果你对无限流有了认识，很多系统的设计都可以看作成一个无限流。比如，一些大数据平台，它就是有源源不断的数据流入其中，而我们要做的就是给这个无限流提供各种转换，你去看看现在炙手可热的Flink，它使用的就是这种思路。

记忆

我们再来看另一个关于惰性求值带来的有趣做法：记忆（Memoization）。

前面说过，Proxy模式之所以要采用惰性求值的策略，一个重要的原因就是真正的计算部分往往是消耗很大的。所以，一旦计算完成，一个好的策略就是将计算的结果缓存起来，这样，再次调用时就不必重新计算了。其实，这种做法就是记忆。

记忆，在Wikipedia上是这样定义的：

> 在计算中，记忆是一种优化技术，主要用于加速计算机程序，其做法就是将昂贵函数的结果存储起来，当使用同样的输入再次调用时，返回其缓存的结果。

这里的一个重点是，同样的输入。我们已经知道了，函数式编程中的函数是纯函数，同样的输入必然会给出同样的输出。所以，我们就不难理解，记忆这种技术在函数式编程中的作用了。

实现记忆这种技术并不难，下面就给出了一个实现，这里用到了Java并发库中的类AtomicReference，从而消除了可能产生的多线程问题：

```
public static <T> Supplier<T> memoize(Supplier<T> delegate) {
  AtomicReference<T> value = new AtomicReference<>();
  return () -> {
    T val = value.get();
    if (val == null) {
      synchronized(value) {
        val = value.get();
        if (val == null) {
          val = Objects.requireNonNull(delegate.get());
          value.set(val);
        }
```

```
      }
    }
    return val;
  };
}
```

这个实现用起来也很简单：

```
long ultimateAnswer = memoize(() -> {
    // 这里有一个常常的计算
    // 返回一个终极答案
    return 42;
})
```

在这里，memoize是一个通用的实现，它的适用范围很广。我们仔细对比就不难发现，这里我们已经实现了Proxy模式的能力，换言之，有了它，我们可以不再需要Proxy模式。后面我们讲到设计模式也会提到，一些设计模式是受限于程序设计语言自身能力不足而出现的，这里也算为这个观点添加了一个注脚。

Optional

让我们回到学生的例子上，如果想获取一个学生出生的国家，我们该怎么写这段代码呢？直觉上的写法是这样的：

```
public Country getBirthCountry() {
  return this.getBirthPlace() // 获取出生地
              .getCity()      // 获取城市
              .getProvince()  // 获取省份
              .getCountry();  // 获取国家
}
```

然而，在真实项目中，代码并不能这么写，因为这样可能会出现空指针，所以，我们不得不把代码写成这样：

```
public Country getBirthCountry() {
  Place place = this.birthPlace;
  if (place != null) {
    City city = place.getCity();
    if (city != null) {
      Province province = city.getProvince();
      if (province != null) {
        return province.getCountry();
      }
    }
  }

  return null;
}
```

这是一段令人作呕的代码，但我们不得不这么写，因为空指针总是一个令人头疼的问题。事实上，作为程序员，我们经常会有忘记做空指针检查的时候。这不是一个人的问题，而是整个行业的问题， IT 行业每年都会因此造成巨大的损失。

> 我将其称为自己犯下的十亿美元错误……
> I call it my billion-dollar mistake…
> ——Sir C. A. R. Hoare，空引用的发明者

难道空指针就是一个无解的问题吗？程序员们并不打算束手就擒，于是，一种新的解决方案产生了，就是可选对象。这个解决方案在Java 8中叫Optional，在Scala中叫Option。接下来，我们就以Java 8中的Optional为例进行讲解。

Optional是一个对象容器，其中可能包含着一个非空的值，也可能不包含。这是什么意思呢？它和直接使用对象的场景是一一对应的，如果包含值，就对应着就是有值的场景；而不包含，则对应着值为空的场景。

那该如何去创建一个Optional对象呢？

- 如果有一个非空对象，可以用 of() 将它包装成一个 Optional 对象；
- 如果要表示空，可以返回一个 empty()；
- 如果有一个从别处传来的对象，你不知道它是不是空，可以用 ofNullable()。

```
Optional.of("Hello"); // 创建一个Optional对象，其中包含了"Hello"字符串
Optional.empty(); // 创建了一个表示空对象的Optional对象。
Optional.ofNullable(instance); // 创建了一个Optional对象，不知instance是否为空。
```

也许你会好奇，直接使用对象都解决不了问题，把对象放到一个容器里就解决了？还真能。因为你要用这个对象的时候，需要把对象取出来，而要取出对象，你就需要判断一下这个对象是否为空。就像下面这面代码这样：

```
if (country.isPresent()) {
  return country.get();
}
```

只有Optional里包含的是一个非空的对象时， get() 方法才能正常执行，否则，就会抛出异常。显然，当你调用 get()的时候，意图是很明显的，我要处理的是一个非空的值，所以，就必须加上一段判断对象是否存在的代码。

这比直接访问对象多用了一步，但正是这多出的一步让你的大脑必须想一下，自己是否需要加上判空的处理，而不是像普通对象一样，一下子就滑了过去。

而且因为 get()本身是有意图的，用工具也可以扫描出缺失的判断，比如，如果你用IntelliJ IDEA写程序的话，不加判断，直接get()的话，它就会给你一个警告。

使用Optional，我们还可以给空对象增加一些额外的处理，比如给个缺省值：

```
country.orElse(china);  // 返回一个缺省的对象
```

也可以生成一个新的对象：

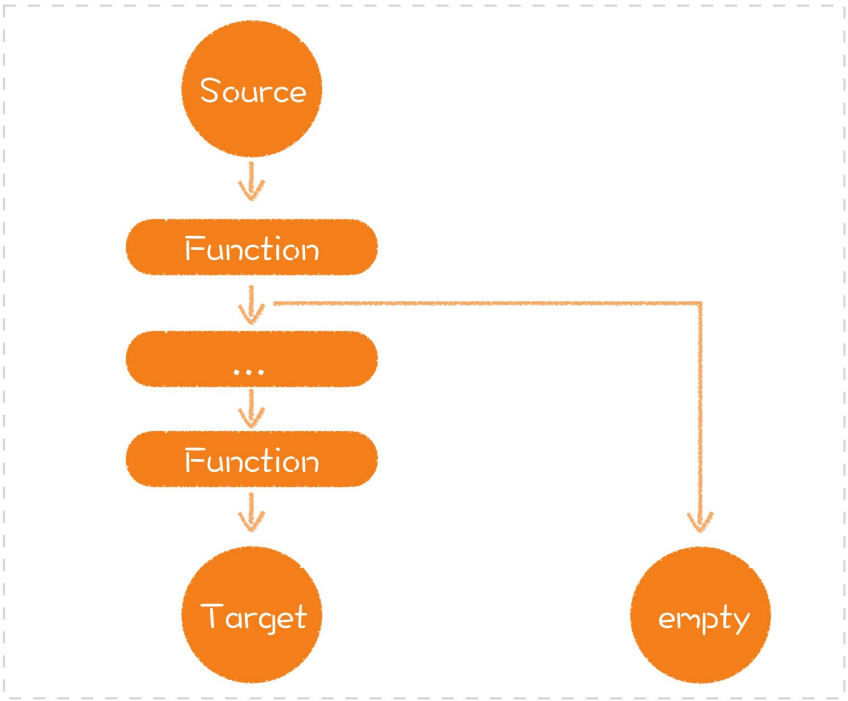
```
country.orElseGet(Country::new); // 调用了一个函数生成了一个新对象
```

或是抛出异常：

```
country.orElseThrow(IllegalArgumentException::new);
```

其实，我们拿到一个值之后，往往要做一些更多的处理。使用了Optional，我们甚至可以不用把其中的值取出来，直接就做一些处理了。比如，它提供map、flatMap、filter等一些方法，就是当Optional包含的对象不为空时，调用对应的方法做处理，为空的时候，直接返回表示空的Optional对象。

从下面这张图，你就能够理解这些方法的基本逻辑：



好，有了对Optional的基本了解，我们在日常工作中怎么用它呢？很简单，**在方法需要返回一个值时，如果返回的对象可能为空，那就返回一个Optional**。这样就给了这个方法使用者一个提示，这个对象可能为

空，小心处理。

比如，获取学生的出生地，方法可以这么写：

```
Optional<Place> getBirthPlace() {
  return Optional.ofNullable(this.birthPlace);
}
```

好，回到我们前面的问题上。获取一个学生出生的国家，代码可以怎么写呢？如果相应的方法都改写成Optional，代码写出来会是这个样子：

```
public Optional<Country> getBirthCountry() {
  return Optional.ofNullable(this.birthPlace)
           .flatMap(Place::getCity)
           .flatMap(City::getProvince)
           .flatMap(Province::getCountry);
}
```

虽然我们不能说这段代码一定有多优雅，但是至少比层层嵌套的if判断要整洁一些了。

最后，你可能会问，这个Optional和函数式编程有什么关系呢？其实，Optional将对象封装起来的做法来自于函数式编程中一个叫Monad的概念，你可以简单地把它理解成一个对象容器。Optional就对应着其中的一种：Maybe Monad。

我们前面也看到了，正是因为这个容器的存在，解决了很多问题。Monad 的概念解释起来还有很多东西要说，篇幅所限，就不过多阐述了，有兴趣不妨自己去了解一下。

这种对象容器的思想也逐渐在开枝散叶，比如，在Rust的标准库里，有一个Result，用来定义可恢复的故障。它可以是一个正常值，也可以是一个错误值：

```
enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

下面是一段摘自Rust标准库文档的代码，当我们有了前面对于Optional的讲解，理解起这段代码也就容易多了。

```
enum Version { Version1, Version2 }

// 定义一个解析版本的函数
fn parse_version(header: &[u8]) -> Result<Version, &'static str> {
```

```
    match header.get(0) {
        None => Err("invalid header length"), // 无法解析，返回错误
        Some(&1) => Ok(Version::Version1),  // 解析出版本1
        Some(&2) => Ok(Version::Version2),  // 解析出版本2
        Some(_) => Err("invalid version"),  // 无效版本，返回错误
    }
}

let version = parse_version(&[1, 2, 3, 4]);
// 根据返回值进行处理
match version {
    Ok(v) => println!("working with version: {:?}", v),
    Err(e) => println!("error parsing header: {:?}", e),
}
```

总结时刻

今天，我给你讲了两个比较有用的函数式编程的概念：惰性求值和Optional。

惰性求值是一种求值策略，它将求值的过程延迟到真正需要这个值的时候，其作用就是规避一些不必要的计算。因为惰性求值的存在，还衍生出一些有趣的做法，比如，无限流和记忆。无限流启发了现在的一些大数据平台的设计，而记忆可以很好地替代Proxy模式。

Optional是为了解决空对象而产生的，它其实就是一个对象容器。因为这个容器的存在，访问对象时，需要增加一步思考，减少犯错的几率。

正如我在前面课程中讲到，函数式编程中有很多优秀的内容，值得我们去学习借鉴。我在这几讲中讲到的内容，也只能说是管中窥豹，帮助你见识函数式编程一些优秀的地方。

如果你想了解更多函数式编程，不妨读读《计算机程序的构造与解释》，体会一层一层构建抽象的美妙。如果还想了解更多，那就找一门函数式编程语言去学习一下。

如果今天的内容你只能记住一件事，那请记住：**花点时间学习函数式编程。**

思考题

现在，你已经对函数式编程不陌生了，我想请你谈谈学习函数式编程的感受，无论你是刚刚跟着我学习的，还是之前已经学习过的，欢迎在留言区分享你的想法。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

精选留言：

- qinsi 2020-07-10 08:17:36
  * 记忆化是Memoization，正文里应该是拼错了，示例代码里是对的；
  * 个人理解单凭记忆化还无法取代Proxy模式，因为Proxy模式主要是做方法调用的分发(dispatch)，在分发时可以做些额外的事情（比如记忆化）。单是实现动态分发的话Java里可以用反射，Ruby里可以用method_missing等，并不是一种很稀缺的语言特性；
  * Option的价值在于类型而非对象。是类型的话在编译时编译器就可以进行检查，而不是依赖程序员在运行时进行检查（或是依赖IDE）。能处理Option的函数也是这个思路，程序员在进行中间处理时可以不用自己处理empty值，只需要确保类型正确；

* Monad是个被诅咒的名字，日常开发中不应该提到它 ;-) [1赞]

作者回复2020-07-10 09:35:00
多谢提醒纠错！

非常好的补充，你的补充提升了这篇文章的整体价值。

你对 Monad 的观点，我非常同意！

- 阳仔 2020-07-10 09:21:33
函数式编程是应当好好普及，这是一个编程思想的转变

- 被雨水过滤的空气 2020-07-10 09:10:31
函数式比较难的是怎么样对外界施加作用，又能保证写的是纯函数。

作者回复2020-07-10 09:35:46
在实际工作中，努力把外部的部分隔离开来。让自己的代码纯起来。